Semantic-head-driven generation

SEMANTIC-HEAD-DRIVEN GENERATION

Stuart M. Shieber
Aiken Computation Laboratory
Division of Applied Sciences
Harvard University
Cambridge, MA 02138

Fernando C. N. Pereira
AT & T Bell Laboratories
Murray Hill, NJ 07974

Gertjan van Noord
Department of Linguistics
Rijksuniversiteit Utrecht
Utrecht, The Netherlands

Robert C. Moore
Artificial Intelligence Center
SRI International
Menlo Park, CA 94025

We present an algorithm for generating strings from logical form encodings that improves upon previous algorithms in that it places fewer restrictions on the class of grammars to which it is applicable. In particular, unlike a previous bottom-up generator, it allows use of semantically nonmonotonic grammars, yet unlike top-down methods, it also permits left-recursion. The enabling design feature of the algorithm is its implicit traversal of the analysis tree for the string being generated in a semantic-head-driven fashion.

1 INTRODUCTION

The problem of generating a well-formed natural language expression from an encoding of its meaning possesses properties that distinguish it from the converse problem of recovering a meaning encoding from a given natural language expression. This much is axiomatic. In previous work (Shieber 1988), however, one of us attempted to characterize these differing properties in such a way that a single uniform architecture, appropriately parameterized, might be used for both natural language processes. In particular, we developed an architecture inspired by the Earley deduction work of Pereira and Warren (1983), but which generalized that work allowing for its use in both a parsing and generation mode merely by setting the values of a small number of parameters.

As a method for generating natural language expressions, the Earley deduction method is reasonably successful along certain dimensions. It is quite simple, general in its applicability to a range of unification-based and logic grammar formalisms, and uniform, in that it places only one restriction (discussed below) on the form of the linguistic analyses allowed by the grammars used in generation. In particular, generation from grammars with recursions whose well-foundedness relies on lexical information will terminate; top-down generation regimes such as those of Wedekind (1988) or Dymetman and Isabelle (1988) lack this property; further discussion can be found in Section 2.1.

Unfortunately, the bottom-up, left-to-right processing regime of Earley generation—as it might be called—has its own inherent frailties. Efficiency considerations require that only grammars possessing a property of semantic monotonicity can be effectively used, and even for those grammars, processing can become overly nondeterministic.

The algorithm described in this paper is an attempt to resolve these problems in a satisfactory manner. Although we believe that this algorithm could be seen as an instance of a uniform architecture for parsing and generation—just as the extended Earley parser (Shieber, 1985b) and the bottom-up generator were instances of the generalized Earley deduction architecture—our efforts to date have been aimed foremost toward the development of the algorithm for generation alone. We will mention efforts toward this end in Section 5.

1.1 APPLICABILITY OF THE ALGORITHM

As does the Earley-based generator, the new algorithm assumes that the grammar is a unification-based or logic grammar with a phrase structure backbone and complex nonterminals. Furthermore, and again consistent with previous work, we assume that the nonterminals associate to the phrases they describe logical expressions encoding their possible meanings. Beyond these requirements common to logic-based formalisms, the methods are generally applicable.

A variant of our method is used in Van Noord's BUG (Bottom-Up Generator) system, part of MiMo2, an experimental machine translation system for translating international news items of Teletext, which uses a Prolog version of

PATR-II similar to that of Hirsh (1987). According to Martin Kay (personal communication), the STREP machine translation project at the Center for the Study of Language and Information uses a version of our algorithm to generate with respect to grammars based on head-driven phrase structure grammar (HPSG). Finally, Calder et al. (1989) report on a generation algorithm for unification categorial grammar that appears to be a special case of ours.

1.2 PRELIMINARIES

Despite the general applicability of the algorithm, we will, for the sake of concreteness, describe it and other generation algorithms in terms of their implementation for definite-clause grammars (DCG). For ease of exposition, the encoding will be a bit more cumbersome than is typically found in Prolog DCG interpreters. The standard DCG encoding in Prolog uses the notation

$$\langle cat_0 \rangle \text{ --> } \langle cat_1 \rangle, \ldots, \langle cat_n \rangle.$$

where the $\langle cat_i \rangle$ are terms representing the grammatical category of an expression and its subconstituents. Terminal symbols are introduced into rules by enclosing them in list brackets, for example

```
sbar/S --> [that], s/S.
```

Such rules can be translated into Prolog directly using a difference list encoding of string positions; we assume readers are familiar with this technique (Pereira and Shieber, 1985).

Because we concentrate on the relationship between expressions in a language and their logical forms, we will assume that the category terms have both a syntactic and a semantic component. In particular, the infix function symbol / will be used to form categories of the form Syn/Sem where Syn is the syntactic category of the expression and Sem is an encoding of its semantics as a logical form; the previous rule uses this notation, for example. From a DCG perspective, all the rules involve the single nonterminal /, with the given intended interpretation. Furthermore, the representation of grammars that we will postulate includes the threading of string positions explicitly, so that a node description will be of the form node (Syn/Sem, PO-P). The first argument of the node functor is the category, divided into its syntactic and semantic components; the second argument is the difference list encoding of the substring it covers. In summary, a DCG grammar rule will be encoded as the clause

$$\begin{aligned}&\text{node}(\langle syn_0 \rangle / \langle sem_0 \rangle, \text{P0-P}) \text{ --->} \\ &\quad [\text{node}(\langle syn_1 \rangle / \langle sem_1 \rangle, \text{P0-P1}), \ldots, \\ &\quad \text{node}(\langle syn_n \rangle / \langle sem_n \rangle), \text{P}_{n-1}\text{-P}].\end{aligned}$$

We use the functor '--->' to distinguish this *node encoding* from the standard one. The right-hand-side elements are kept as a Prolog list for easier manipulation by the interpreters we will build.

We turn now to the issue of terminal symbols on the right-hand sides of rules in the node encoding. During the compilation process from the standard encoding to the node encoding, the right-hand side of a rule is converted from a list of categories and terminal strings to a list of nodes connected together by the difference-list threading technique used for standard DCG compilation. At that point, terminal strings can be introduced into the string threading and need never be considered further. For instance, the previous rule becomes

```
node(sbar/S, [that|P0]-P) ---> node(s/S, P0-P).
```

Throughout, we will alternate between the two encodings, using the standard one for readability and the node encoding as the actual data for grammar interpretation. As the latter, more cumbersome, representation is algorithmically generable from the former, no loss of generality ensues from using both.

2 PROBLEMS WITH EXISTING GENERATORS

Existing generation algorithms have efficiency or termination problems with respect to certain classes of grammars. We review the problems of both top-down and bottom-up regimes in this section.

2.1 PROBLEMS WITH TOP-DOWN GENERATORS

Consider a naive top-down generation mechanism that takes as input the semantics to generate from and a corresponding syntactic category and builds a complete tree, top-down, left-to-right by applying rules of the grammar nondeterministically to the fringe of the expanding tree. This control regime is realized, for instance, when running a DCG "backwards" as a generator.

Concretely, the following DCG interpreter—written in Prolog and taking as its data the grammar in encoded form—implements such a generation method.

```
gen(LF, Sentence) :- generate(node(s/LF, Sentence-[ ])).
generate(Node) :-
    (Node ---> Children),
    generate_children(Children).
generate_children([ ]).
generate_children([Child|Rest]) :-
    generate(Child),
    generate_children(Rest).
```

Clearly, such a generator may not terminate. For example, consider a grammar that includes the rules

```
s/S --> np/NP, vp(NP)/S.
np/NP --> det(N)/NP, n/N.
det(N)/NP --> np/NP0, poss(NP0,N)/NP.
np/john --> [john].
poss(NP0,N)/mod(N,NP0) --> [s].
n/father --> [father].
vp(NP)/left(NP) --> [left].
```

This grammar admits sentences like "John left" and "John's father left" with logical form encodings left(john) and left(mod(father, john)), respectively. The technique used here to build the logical forms is well-known in logic grammars.[1]

Generation with the goal gen(left(john), Sent) using the generator above will result in application of the first rule to the node node(s/left(john), Sent-[ ]). A subgoal for the generation of a node node(np/NP, Sent-P) will result. To this subgoal, the second rule will apply, leading to a subgoal for generation of the node node(det(N)/NP, Sent-P1), which itself, by virtue of the third rule, leads to another instance of the NP node generation subgoal. Of course, the loop may now be repeated an arbitrary number of times. Graphing the tree being constructed by the traversal of this algorithm, as in Figure 1, immediately exhibits the potential for nontermination in the control structure. (The repeated goals along the left branch are presented in boldface in the figure. Dashed lines indicate portions of the tree yet to be generated.)

This is an instance of the general problem familiar from logic programming that a logic program may not terminate when called with a goal less instantiated than what was intended by the program's designer. Several researchers have noted that a different ordering of the branches in the top-down traversal would, in the case at hand, remedy the nontermination problem. For the example above, the solution is to generate the VP first—using the goal generate (node(vp(NP)/left(john), P1-[ ]))—in the course of which the variable NP will become bound so that the generation from node(np/NP, Sent-P1) will terminate.

We might allow for reordering of the traversal of the children by sorting the nodes before generating them. This can be simply done, by modifying the first clause of generate.

```
generate(Node) :-
    (Node ---> Children),
    sort_children(Children, SortedChildren),
    generate_children(SortedChildren).
```

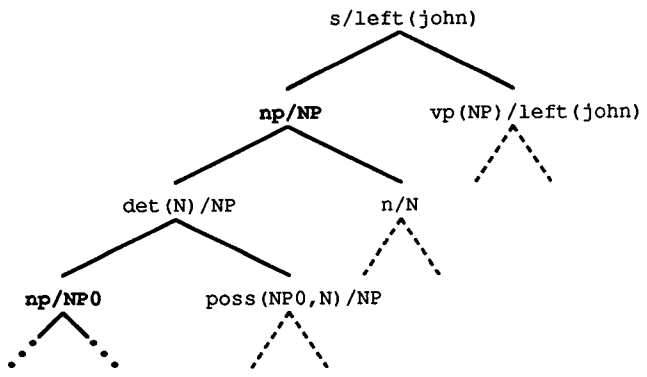

Figure 1 Tree Constructed Top-Down by Left-Recursive Grammar.

Here, we have introduced a predicate sort_children to reorder the child nodes before generating. Dymetman and Isabelle (1988) propose a node-ordering solution to the top-down nontermination problem; they allow the grammar writer to specify a separate goal ordering for parsing and for generation by annotating the rules by hand. Strzalkowski (1989) develops an algorithm for generating such annotations automatically. In both of these cases, the node ordering is known a priori, and can be thought of as applying to the rules at compile time.

Wedekind (1988) achieves the reordering by first generating nodes that are *connected*, that is, whose semantics is instantiated. Since the NP is not connected in this sense, but the VP is, the latter will be expanded first. In essence, the technique is a kind of goal freezing (Colmerauer 1982) or implicit *wait* declaration (Naish 1986). This method is more general, as the reordering is dynamic; the ordering of child nodes can, in principle at least, be different for different uses of the same rule. The generality seems necessary; for cases in which the a priori ordering of goals is insufficient, Dymetman and Isabelle also introduce goal freezing to control expansion.

Although vastly superior to the naive top-down algorithm, even this sort of amended top-down approach to generation based on goal freezing under one guise or another is insufficient with respect to certain linguistically plausible analyses. The symptom is an ordering paradox in the sorting. For example, the "complements" rule given by Shieber (1985a) in the PATR-II formalism

$$\begin{aligned} VP_1 \rightarrow\ & VP_2\ X \\ & \langle VP_1\ \text{head} \rangle = \langle VP_2\ \text{head} \rangle \\ & \langle VP_2\ \text{syncat first} \rangle = \langle X \rangle \\ & \langle VP_2\ \text{syncat rest} \rangle = \langle VP_1\ \text{syncat} \rangle \end{aligned}$$

can be encoded as the DCG rule:

```
vp(Head, Syncat)/VP -->
    vp(Head, [Compl/LF|Syncat])/VP,    Compl/LF.
```

Top-down generation using this rule will be forced to expand the lower VP before its complement, since LF is uninstantiated initially. Any of the reordering methods must choose to expand the child VP node first. But in that case, application of the rule can recur indefinitely, leading to nontermination. Thus, no matter what ordering of subgoals is chosen, nontermination results.

Of course, if one knew ahead of time that the subcategorization list being built up as the value for Syncat was bounded in size, then an ad hoc solution would be to limit recursive use of this rule when that limit had been reached. But even this ad hoc solution is problematic, as there may be no principled bound on the size of the subcategorization list. For instance, in analyses of Dutch cross-serial verb constructions (Evers 1975; Huybrechts 1984), subcategorization lists may be concatenated by syntactic rules (Moort-

gat 1984; Fodor et al. 1985; Pollard 1988), resulting in arbitrarily long lists. Consider the Dutch sentence

dat [Jan [Marie [de oppasser [de olifanten [zag helpen
that John Mary the keeper the elephants saw help

voeren]]]]
feed

that John saw Mary help the keeper feed the elephants

The string of verbs is analyzed by appending their subcategorization lists as in Figure 2. Subcategorization lists under this analysis can have any length, and it is impossible to predict from a semantic structure the size of its corresponding subcategorization list merely by examining the lexicon.

Strzalkowski refers to this problem quite aptly as constituting a deadlock situation. He notes that by combining deadlock-prone rules (using a technique akin to partial execution[2]) many deadlock-prone rules can be replaced by rules that allow reordering; however, he states that "the general solution to this normalization problem is still under investigation." We think that such a general solution is unlikely because of cases like the one above in which no finite amount of partial execution can necessarily bring sufficient information to bear on the rule to allow ordering. The rule would have to be partially executed with respect to itself and all verbs so as to bring the lexical information that well-founds the ordering to bear on the ordering problem. In general, this is not a finite process, as the previous Dutch example reveals. This does not deny that compilation methods may be able to convert a grammar into a program that generates without termination problems. In fact, the partial execution techniques described by two of us (Pereira and Shieber 1985) could form the basis of a compiler built by partial execution of the new algorithm we propose below relative to a grammar. However, the compiler will not generate a program that generates top-down, as Strzalkowski's does.

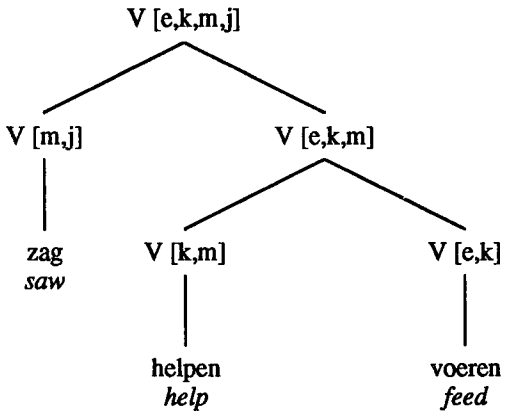

Figure 2 Schematic of Verb Subcategorization Lists for Dutch Example.

In summary, top-down generation algorithms, even if controlled by the instantiation status of goals, can fail to terminate on certain grammars. The critical property of the example given above is that the well-foundedness of the generation process resides in lexical information unavailable to top-down regimes. This property is the hallmark of several linguistically reasonable analyses based on lexical encoding of grammatical information such as are found in categorial grammar and its unification-based and combinatorial variants, in head-driven phrase-structure grammar, and in lexical-functional grammar.

2.2 PROBLEMS WITH BOTTOM-UP GENERATORS

The bottom-up Earley-deduction generator does not fall prey to these problems of nontermination in the face of recursion, because lexical information is available immediately. However, several important frailties of the Earley generation method were noted, even in the earlier work.

For efficiency, generation using this Earley deduction method requires an incomplete search strategy, filtering the search space using semantic information. The semantic filter makes generation from a logical form computationally feasible, but preserves completeness of the generation process only in the case of semantically monotonic grammars—those grammars in which the semantic component of each right-hand-side nonterminal subsumes some portion of the semantic component of the left-hand-side. The semantic monotonicity constraint itself is quite restrictive. As stated in the original Earley generation paper (Shieber 1988), "perhaps the most immediate problem raised by [Earley generation] is the strong requirement of semantic monotonicity.... Finding a weaker constraint on grammars that still allows efficient processing is thus an important research objective." Although it is intuitively plausible that the semantic content of subconstituents ought to play a role in the semantics of their combination—this is just a kind of compositionality claim—there are certain cases in which reasonable linguistic analyses might violate this intuition. In general, these cases arise when a particular lexical item is stipulated to occur, the stipulation being either lexical (as in the case of particles or idioms) or grammatical (as in the case of expletive expressions).

Second, the left-to-right scheduling of Earley parsing, geared as it is toward the structure of the string rather than that of its meaning, is inherently more appropriate for parsing than generation.[3] This manifests itself in an overly high degree of nondeterminism in the generation process. For instance, various nondeterministic possibilities for generating a noun phrase (using different cases, say) might be entertained merely because the NP occurs before the verb which would more fully specify, and therefore limit, the options. This nondeterminism has been observed in practice.

2.3 SOURCE OF THE PROBLEMS

We can think of a parsing or generation process as *discovering* an analysis tree,[4] one admitted by the grammar and

satisfying certain syntactic or semantic conditions, by traversing a virtual tree and constructing the actual tree during the traversal. The conditions to be satisfied—possessing a given yield in the parsing case, or having a root node labeled with given semantic information in the case of generation—reflect the different premises of the two types of problems. This perspective purposely abstracts issues of nondeterminism in the parsing or generation process, as it assumes an oracle to provide traversal steps that happen to match the ethereal virtual tree being constructed. It is this abstraction that makes it a useful expository device, but should not be taken literally as a description of an algorithm.

From this point of view, a naive top-down parser or generator performs a depth-first, left-to-right traversal of the tree. Completion steps in Earley's algorithm, whether used for parsing or generation, correspond to a post-order traversal (with prediction acting as a pre-order filter). The left-to-right traversal order of both of these methods is geared towards the given information in a parsing problem, the string, rather than that of a generation problem, the goal logical form. It is exactly this mismatch between structure of the traversal and structure of the problem premise that accounts for the profligacy of these approaches when used for generation.

Thus, for generation, we want a traversal order geared to the premise of the generation problem, that is, to the semantic structure of the sentence. The new algorithm is designed to reflect such a traversal strategy respecting the semantic structure of the string being generated, rather than the string itself.

3 The New Algorithm

Given an analysis tree for a sentence, we define the *pivot node* as the lowest node in the tree such that it and all higher nodes up to the root have the same semantics. Intuitively speaking, the pivot serves as the *semantic head* of the root node. Our traversal will proceed both top-down and bottom-up from the pivot, a sort of semantic-head-driven traversal of the tree. The choice of this traversal allows a great reduction in the search for rules used to build the analysis tree.

To be able to identify possible pivots, we distinguish a subset of the rules of the grammar, the *chain rules,* in which the semantics of some right-hand-side element is identical to the semantics of the left-hand-side. The right-hand-side element will be called the rule's semantic head. The traversal, then, will work top-down from the pivot using a nonchain rule, for if a chain rule were used, the pivot would not be the lowest node sharing semantics with the root. Instead, the pivot's semantic head would be. After the nonchain rule is chosen, each of its children must be generated recursively.

The bottom-up steps to connect the pivot to the root of the analysis tree can be restricted to chain rules only, as the pivot (along with all intermediate nodes) has the same semantics as the root and must therefore be the semantic head. Again, after a chain rule is chosen to move up one node in the tree being constructed, the remaining (non-semantic-head) children must be generated recursively.

The top-down base case occurs when the nonchain rule has no nonterminal children; that is, it introduces lexical material only. The bottom-up base case occurs when the pivot and root are trivially connected because they are one and the same node.

An interesting side issue arises when there are two right-hand-side elements that are semantically identical to the left-hand-side. This provides some freedom in choosing the semantic head, although the choice is not without ramifications. For instance, in some analyses of NP structure, a rule such as

```
np/NP --> det/NP, nbar/NP.
```

is postulated. In general, a chain rule is used bottom-up from its semantic head and top-down on the non-semantic-head siblings. Thus, if a non-semantic-head subconstituent has the same semantics as the left-hand-side, a recursive top-down generation with the same semantics will be invoked. In theory, this can lead to nontermination, unless syntactic factors eliminate the recursion, as they would in the rule above regardless of which element is chosen as semantic head. In a rule for relative clause introduction such as the following (in highly abbreviated form)

```
nbar/N --> nbar/N, sbar/N.
```

we can (and must) choose the nominal as semantic head to effect termination. However, there are other problematic cases, such as verb-movement analyses of verb-second languages. We discuss this topic further in Section 4.3.

3.1 A DCG Implementation

To make the description more explicit, we will develop a Prolog implementation of the algorithm for DCGs, along the way introducing some niceties of the algorithm previously glossed over.

As before, a term of the form node(Cat, P0-P) represents a phrase with the syntactic and semantic information given by Cat starting at position P0 and ending at position P in the string being generated. As usual for DCGs, a string position is represented by the list of string elements after the position. The generation process starts with a goal category and attempts to generate an appropriate node, in the process instantiating the generated string.

```
gen(Cat, String) :- generate(node(Cat, String-[ ])).
```

To generate from a node, we nondeterministically choose a nonchain rule whose left-hand-side will serve as the pivot. For each right-hand-side element, we recursively generate, and then connect the pivot to the root.

```
generate(Root) :-
    % choose nonchain rule
    applicable_non_chain_rule(Root, Pivot, RHS),
    % generate all subconstituents
    generate_rhs(RHS),
    % generate material on path to root
    connect(Pivot, Root).
```

The processing within generate_rhs is a simple iteration.

```
generate_rhs([ ]).

generate_rhs([First | Rest]) :-
  generate(First),
  generate_rhs(Rest).
```

The connection of a pivot to the root, as noted before, requires choice of a chain rule whose semantic head matches the pivot, and the recursive generation of the remainder of its right-hand side. We assume a predicate applicable_chain_rule(SemHead, LHS, Root, RHS) that holds if there is a chain rule admitting a node LHS as the left-hand side, SemHead as its semantic head, and RHS as the remaining right-hand-side nodes, such that the left-hand-side node and the root node Root can themselves be connected.

```
connect(Pivot, Root) :-
    % choose chain rule
    applicable_chain_rule(Pivot, LHS, Root, RHS),
    % generate remaining siblings
    generate_rhs(RHS),
    % connect the new parent to the root
    connect(LHS, Root).
```

The base case occurs when the root and the pivot are the same. To implement the generator correctly, identity checks like this one must use a sound unification algorithm with the occurs check. (The default unification in most Prolog systems is unsound in this respect.) The reason is simple. Consider, for example, a grammar with a gap-threading treatment of wh-movement (Pereira 1981; Pereira and Shieber 1985), which might include the rule

```
np(Agr, [np(Agr)/Sem|X]-X)/Sem ---> [ ].
```

stating that an NP with agreement Agr and semantics Sem can be empty provided that the list of gaps in the NP can be represented as the difference list [np(Agr)/Sem|X]-X, that is, the list containing an NP gap with the same agreement features Agr. Because the above rule is a nonchain rule, it will be considered when trying to generate any nongap NP, such as the proper noun np(3-sing,G-G)/john. The base case of connect will try to unify that term with the head of the rule above, leading to the attempted unification of X with [np(Agr)/Sem|X], an occurs-check failure that would not be caught by the default Prolog unification algorithm. The base case, incorporating the explicit call to a sound unification algorithm, is therefore as follows:

```
connect(Pivot, Root) :-
    % trivially connect pivot to root
    unify(Pivot, Root).
```

Now, we need only define the notion of an applicable chain or nonchain rule. A nonchain rule is applicable if the semantics of the left-hand side of the rule (which is to become the pivot) matches that of the root. Further, we require a top-down check that syntactically the pivot can serve as the semantic head of the root. For this purpose, we assume a predicate chained_nodes that codifies the transitive closure of the semantic head relation over categories. This is the correlate of the link relation used in left-corner parsers with top-down filtering; we direct the reader to the discussion by Matsumoto et al. (1983) or Pereira and Shieber (1985) for further information.

```
applicable_non_chain_rule(Root, Pivot, RHS) :-
    % semantics of root and pivot are same
    node_semantics(Root, Sem),
    node_semantics(Pivot, Sem),
    % choose a nonchain rule
    non_chain_rule(LHS, RHS),
    % . . . whose lhs matches the pivot
    unify(Pivot, LHS),
    % make sure the categories can connect
    chained_nodes(Pivot, Root).
```

A chain rule is applicable to connect a pivot to a root if the pivot can serve as the semantic head of the rule and the left-hand side of the rule is appropriate for linking to the root.

```
applicable_chain_rule(Pivot, Parent, Root, RHS) :-
    % choose a chain rule
    chain_rule(Parent, RHS, SemHead),
    % . . . whose sem. head matches pivot
    unify(Pivot, SemHead),
    % make sure the categories can connect
    chained_nodes(Parent, Root).
```

The information needed to guide the generation (given as the predicates chain_rule, non_chain_rule, and chained_nodes) can be computed automatically from the grammar. A program to compile a DCG into these tables has in fact been implemented. The details of the process will not be discussed further; interested readers may write to the first author for the required Prolog code.

3.2 A SIMPLE EXAMPLE

We turn now to a simple example to give a sense of the order of processing pursued by this generation algorithm. As in previous examples, the grammar fragment in Figure 3 uses the infix operator / to separate syntactic and semantic category information, and subcategorization for complements is performed lexically.

Consider the generation from the category sentence/decl(call_up(john,friends)). The analysis tree that we will be implicitly traversing in the course of generation is given

```
sentence/decl(S) ---> s(finite)/S.                                    (1)
sentence/imp(S) ---> vp(nonfinite,[np(_)/you])/S.
s(form)/S ---> Subj, vp(Form,[Subj])/S.                               (2)
vp(Form,Subcat)/S --->
    vp(Form,[Compl|Subcat])/S, Compl.                                 (3)
vp(Form,[Subj])/S ---> vp(Form,[Subj])/VP,
adv(VP)/S.
...
vp(finite,[np(_)/O,np(3-sing)/S])/love(S,O) ---> [loves].
...
vp(finite,[np(_)/O,p/up,np(3-sing)/S])/call_up(S,O) --->
    [calls].                                                          (4)
...
vp(finite,[np(3-sing)/S])/leave(S) ---> [leaves].
...
np(3-sing)/john ---> [john].                                          (5)
np(3-pl)/friends ---> [friends].                                      (6)
...
adv(VP)/often(VP) ---> [often].
...
det(3-sing,X,P)/qterm(every,X,P) ---> [every].
...
n(3-sing,X)/friend(X) ---> [friend].
```

Figure 3 Grammar Fragment for Simple Example.

in Figure 4. The rule numbers are keyed to the grammar. The pivots chosen during generation and the branches corresponding to the semantic head relation are shown in boldface.

We begin by attempting to find a nonchain rule that will define the pivot. This is a rule whose left-hand-side semantics matches the root semantics decl(call_up(john, friends)) (although its syntax may differ). In fact, the only such nonchain rule is

```
sentence/decl(S) ---> s(finite)/S.                                    (1)
```

We conjecture that the pivot is labeled sentence/decl(call_up(john,friends)). In terms of the tree traversal, we are implicitly choosing the root node [a] as the pivot. We recursively generate from the child's node [b], whose category is s(finite)/call_up(john, friends). For this category, the pivot (which will turn out to be node [f]) will be defined by the nonchain rule

```
vp(finite,[np(_)/O,p/up,np(3-sing)/S]/call_up(S,O) ---> [calls].(4)
```

(If there were other forms of the verb, these would be potential candidates, but most would be eliminated by the chained_nodes check, as the semantic head relation requires identity of the verb form of a sentence and its VP head. See Section 4.2 for a technique for further reducing the nondeterminism in lexical item selection.) Again, we recursively generate for all the nonterminal elements of the right-hand side of this rule, of which there are none.

We must therefore connect the pivot [f] to the root [b]. A chain rule whose semantic head matches the pivot must be chosen. The only choice is the rule

```
vp(Form,Subcat)/S ---> vp(Form,[Compl|Subcat])/S, Compl.              (3)
```

Unifying the pivot in, we find that we must recursively generate the remaining RHS element np(_)/friends, and then connect the left-hand-side node [e] with category

```
vp(finite,[lex/up, np(3-sing)/john])/call_up(john,friends)
```

to the same root [b]. The recursive generation yields a node covering the string "friends" following the previously generated string "calls". The recursive connection will use the same chain rule, generating the particle "up", and the new node to be connected [d]. This node requires the chain rule

```
s(Form)/S ---> Subj, vp(Form,[Subj])/S.                               (2)
```

for connection. Again, the recursive generation for the subject yields the string "John", and the new node to be connected s(finite)/call_up(john,friends). This last node connects to the root [b] by virtue of identity.

This completes the process of generating top-down from the original pivot sentence/decl(call_up(john,friends)). All that remains is to connect this pivot to the original root. Again, the process is trivial, by virtue of the base case for connection. The generation process is thus completed, yielding the string "John calls friends up". The drawing in Figure 4 summarizes the generation process by showing which steps were performed top-down or bottom-up by arrows on the analysis tree branches.

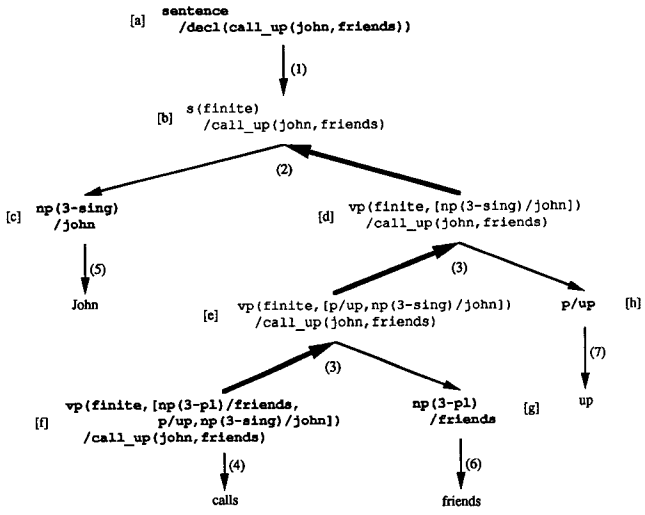

Figure 4 Analysis Tree for Simple Example.

3.3 IMPORTANT PROPERTIES OF THE ALGORITHM

The grammar presented here was forced for expository reasons to be trivial. (We have developed more extensive experimental grammars that can generate relative clauses with gaps and sentences with quantified NPs from quanti-

fied logical forms by using a version of Cooper storage [Cooper, 1983]. An outline of our treatment of quantification is provided in Section 3.4.) Nonetheless, several important properties of the algorithm are exhibited even in the preceding simple example.

First, the order of processing is not left-to-right. The verb was generated before any of its complements. Because of this, full information about the subject, including agreement information, was available before it was generated. Thus, the nondeterminism that is an artifact of left-to-right processing, and a source of inefficiency in the Earley generator, is eliminated. Indeed, the example here was completely deterministic; all rule choices were forced.

In addition, the semantic information about the particle "up" was available, even though this information appears nowhere in the goal semantics. That is, the generator operated appropriately despite a semantically nonmonotonic grammar.

Finally, even though much of the processing is top-down, left-recursive rules, even deadlock-prone rules (e.g. rule (3)), are handled in a constrained manner by the algorithm.

For these reasons, we feel that the semantic-head-driven algorithm is a significant improvement over top-down methods and the previous bottom-up method based on Earley deduction.

3.4 A MORE COMPLEX EXAMPLE: QUANTIFIER STORAGE

We will outline here how the new algorithm can generate, from a quantified logical form, sentences with quantified NPs one of whose readings is the original logical form; that is, how it performs quantifier lowering automatically. For this, we will associate a quantifier store with certain categories and add to the grammar suitable store manipulation rules.

Each category whose constituents may create store elements will have a store feature. Furthermore, for each such category whose semantics can be the scope of a quantifier, there will be an optional nonchain rule to take the top element of an ordered store and apply it to the semantics of the category. For example, here is the rule for sentences:

```
s(Form, GO-G, Store)/quant(Q,X,R,S) --->                    (8)
    s(Form, GO-G, [qterm(Q,X,R)|Store])/S.
```

The term quant(Q,X,R,S) represents a quantified formula with quantifier Q, bound variable X, restriction R, and scope S; qterm(Q,X,R) is the corresponding store element.

In addition, some mechanism is needed to combine the stores of the immediate constituents of a phrase into a store for the phrase. For example, the combination of subject and complement stores for a verb into a clause store is done in one of our test grammars by lexical rules such as

```
vp(finite, [np(_, SO)/O, np(3-sing, SS)/S], SC)/gen(S,O) --->   (9)
    [generates], {shuffle(SS, SO, SC)}.
```

which states that the store SC of a clause with main verb "love" and the stores SS and SO of the subject and object the verb subcategorizes for satisfy the constraint shuffle (SS, SO, SC), meaning that SC is an interleaving of elements of SS and SO in their original order.[5] Constraints in grammar rules such as the one above are handled in the generator by the clause

```
generate({Goals}) :-  call(Goals).
```

which passes the conditions to Prolog for execution. This extension must be used with great care, because it is in general difficult to know the instantion state of such goals when they are called from the generator, and as noted before underinstantiated goals may lead to nontermination. A safer scheme would rely on delaying the execution of goals until their required instantiation patterns are satisfied (Naish 1986).

Finally, it is necessary to deal with the noun phrases that create store elements. Ignoring the issue of how to treat quantifiers from within complex noun phrases, we need lexical rules for determiners, of the form

```
det(3-sing,X,P,[qterm(every,X,P)])/X ---> [every].          (10)
```

stating that the semantics of a quantified NP is simply the variable bound by the store element arising from the NP. For rules of this form to work properly, it is essential that distinct bound logical-form variables be represented as distinct constants in the terms encoding the logical forms. This is an instance of the problem of coherence discussed in Section 4.1.

Figure 5 shows the analysis tree traversal for generating the sentence "No program generates every sentence" from the logical form

```
decl(quant(no,p,prog(p),
           quant(every,s,sent(s),gen(p,s))))
```

The numbers labeling nodes in the figure correspond to tree traversal order. We will only discuss the aspects of the traversal involving the new grammer rules given above. The remaining rules are like the ones in Figure 3, except that nonterminals have an additional store argument where necessary.

Pivot nodes [b] and [c] result from the application of rule (8) to reverse the unstoring of the quantifiers in the goal logical form. The next pivot node is node [j], where rule (9) is applied. For the application of this rule to terminate, it is necessary that at least either the first two or the last argument of the shuffle condition be instantiated. The pivot node must obtain the required store instantiation from the goal node being generated. This happens automatically in the rule applicability check that identified the pivot, since the table chained_nodes identifies the store variables for the goal and pivot nodes. Given the sentence store, the shuffle predicate nondeterministically generates

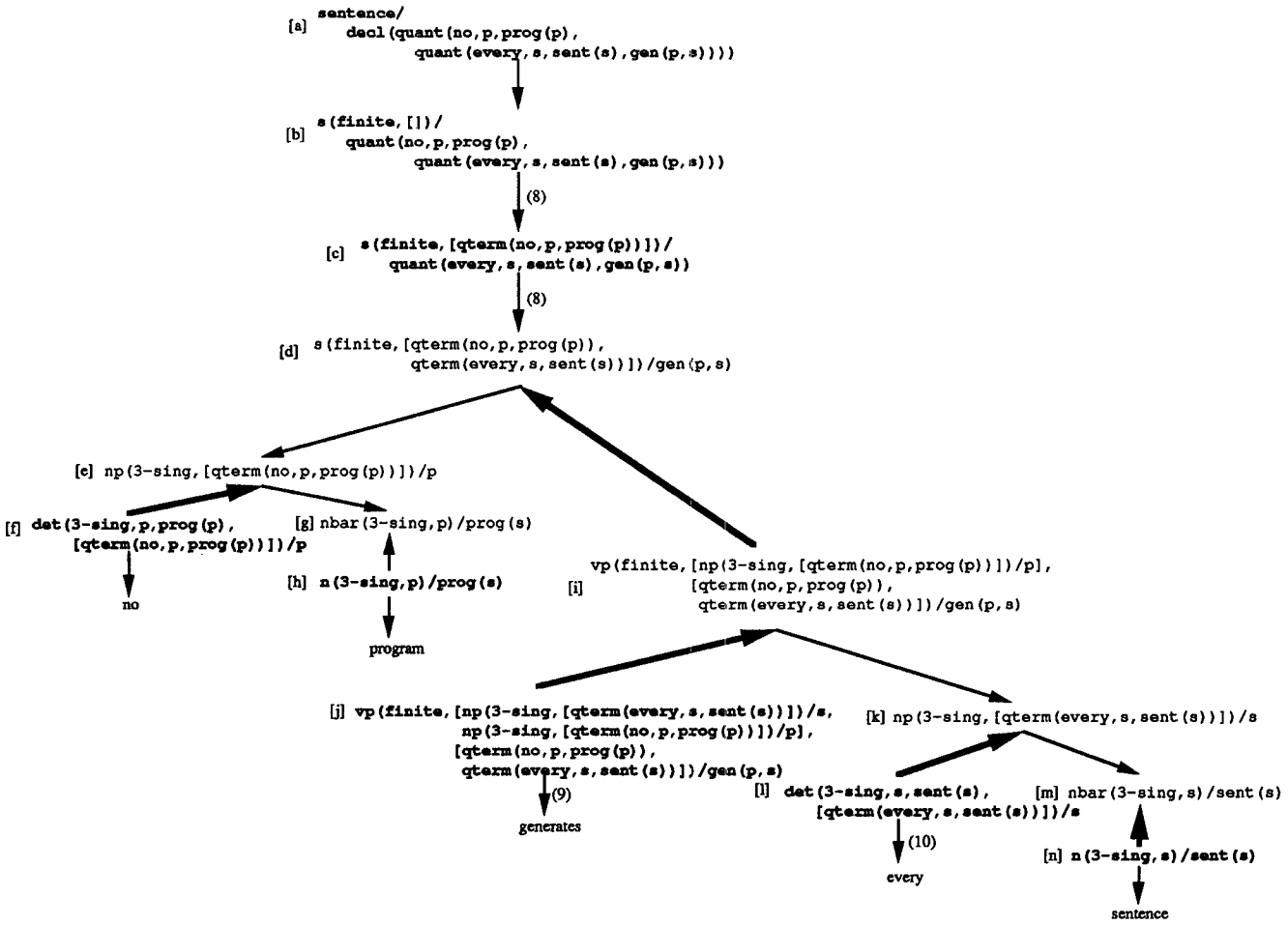

Figure 5 Analysis Tree for Sentence with Quantifiers.

the substores for the constituents subcategorized for by the verb.

The next interesting event occurs at pivot node [l], where rule (10) is used to absorb the store for the object quantified noun phrase. The bound variable for the stored quantifier, in this case s, must be the same as the meaning of the noun phrase and determiner.[6] This condition was already used to filter out inappropriate shuffle results when node [l] was selected as pivot for a noun phrase goal, again through the nonterminal argument identifications included in the chained_nodes table.

The rules outlined here are less efficient than they might be because during the distribution of store elements among the subject and complements of a verb no check is performed as to whether the variable bound by a store element actually appears in the semantics of the phrase to which it is being assigned, leading to many dead ends in the generation process. Also, the rules are sound for generation but not for analysis, because they do not enforce the constraint that every occurrence of a variable in logical form be outscoped by the variable's binder. Adding appropriate side conditions to the rules, following the constraints discussed by Hobbs and Shieber (1987) would not be difficult.

4 Extensions

The basic semantic-head-driven generation algorithm can be augmented in various ways so as to encompass some important analyses and constraints. In particular, we discuss the incorporation of

- completeness and coherence constraints,
- the postponing of lexical choice, and
- the ability to handle certain problematic empty-headed phrases

4.1 Completeness and Coherence

Wedekind (1988) defines completeness and coherence of a generation algorithm as follows. Suppose a generator derives a string w from a logical form s, and the grammar assigns to w the logical form a. The generator is complete if s always subsumes a and coherent if a always subsumes s. The generator defined in Section 3.1 is not coherent or complete in this sense; it requires only that a and s be compatible, that is, unifiable.

If the logical-form language and semantic interpretation system provide a sound treatment of variable binding and

scope, abstraction and application, then completeness and coherence will be irrelevant because the logical form of any phrase will not contain free variables. However, neither semantic projections in lexical-functional grammar (LFG; Halvorsen and Kaplan 1988) nor definite-clause grammars provide the means for such a sound treatment: logical-form variables or missing arguments of predicates are both encoded as unbound variables (attributes with unspecified values in the LFG semantic projection) at the description level. Under such conditions, completeness and coherence become important. For example, suppose a grammar associated the following strings and logical forms.

```
eat(john, X)
'John ate'
eat(john, banana)
'John ate a banana'
eat(john, nice(yellow(banana)))
'John ate a nice yellow banana'
```

The generator of Section 3.1 would generate any of these sentences for the logical form eat(john, X) (because of its incoherence) and would generate "John ate" for the logical form eat(john, banana) (because of its incompleteness).

Coherence can be achieved by removing the confusion between object-level and metalevel variables mentioned above; that is, by treating logical-form variables as constants at the description level. In practice, this can be achieved by replacing each variable in the semantics from which we are generating by a new distinct constant (for instance with the numbervars predicate built into some implementations of Prolog). These new constants will not unify with any augmentations to the semantics. A suitable modification of our generator would be

```
gen(Cat, String) :-
    cat_semantics(Cat,Sem),
    numbervars(Sem,0,_),
    generate(node(Cat,String,[ ])).
```

This leaves us with the completeness problem. This problem arises when there are phrases whose semantics are not ground at the description level, but instead subsume the goal logical form or generation. For instance, in our hypothetical example, the string "John eats" will be generated for semantics eat(john, banana). The solution is to test at the end of the generation procedure whether the feature structure that is found is complete with respect to the original feature structure. However, because of the way in which top-down information is used, it is unclear what semantic information is derived by the rules themselves, and what semantic information is available because of unifications with the original semantics. For this reason, "shadow" variables are added to the generator that represent the feature structure derived by the grammar itself. Furthermore, a copy of the semantics of the original feature structure is made at the start of the generation process. Completeness is achieved by testing whether the semantics of the shadow is subsumed by the copy.

4.2 POSTPONING LEXICAL CHOICE

As it stands, the generation algorithm chooses particular lexical forms on-line. This approach can lead to a certain amount of unnecessary nondeterminism. The choice of a particular form depends on the available semantic and syntactic information. Sometimes there is not enough information available to choose a form deterministically. For instance, the choice of verb form might depend on syntactic features of the verb's subject available only after the subject has been generated. This nondeterminism can be eliminated by deferring lexical choice to a postprocess. Inflectional and orthographical rules are only applied when the generation process is finished and all syntactic features are known. In short, the generator will yield a list of lexical items instead of a list of words. To this list the inflectional and orthographical rules are applied.

The MiMo2 system incorporates such a mechanism into the previous generation algorithm quite successfully. Experiments with particular grammars of Dutch, Spanish, and English have shown that the delay mechanism results in a generator that is faster by a factor of two or three on short sentences. Of course, the same mechanism could be added to any of the other generation techniques discussed in this paper; it is independent of the traversal order.

The particular approach to delaying lexical choice found in the MiMo2 system relies on the structure of the system's morphological component as presented in Figure 6. The figure shows how inflectional rules, orthographical rules, morphology and syntax are related: orthographical rules are applied to the results of inflectional rules. These infectional rules are applied to the results of the morphological rules. The result of the orthographical part are then input for the syntax.

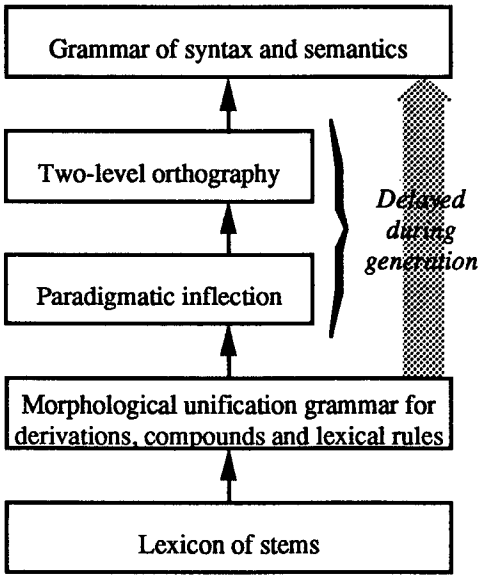

Figure 6 Relation between Morphological Components for Lexical Choice Delaying.

However, in the lexical-delayed scheme the inflectional and orthographical rules are delayed. During the generation process the results of the morphological grammar are used directly. We emphasize that this is possible only because the inflectional and orthographical rules are monotonic, in the sense that they only further instantiate the feature structure of a lexical item but do not change it. This implies, for example, that a rule that relates an active and a passive variant of a verb will not be an inflectional rule but rather a rule in the morphological grammar, although the rule that builds a participle from a stem may in fact be an inflectional rule if it only instantiates the feature vform. When the generation process proper is finished the delayed rules are applied and the correct forms can be chosen deterministically.

The delay mechanism is useful in the following two general cases:

First, the mechanism is useful if an inflectional variant depends on syntatic features that are not yet available. The particular choice of whether a verb has singular or plural inflection depends on the syntactic agreement features of its subject; these are only available after the subject has been generated. Other examples may include the particular choice of personal and relative pronouns, and so forth.

Second, delaying lexical choice is useful when there are several variants for some word that are equally possible because they are semantically and syntactically identical. For example, a word may have several spelling variants. If we delay orthography then the generation process computes with only one "abstract" variant. After the generation process is completed, several variants can be filled in for this abstract one. Examples from English include words that take both regular and irregular tense forms (e.g. "burned/burnt"); and variants such as "traveller/traveler," realize/realise," etc.

4.3 EMPTY HEADS

The success of the generation algorithm presented here comes about because lexical information is available as soon as possible. Returning to the Dutch examples in Section 2.1, the list of subcategorization elements is usually known in time. Semantic heads can then deterministically pick out their arguments.

An example in which this is not the case is an analysis of German and Dutch, where the position of the verb in root sentences (the second position) is different from its position in subordinates (the last position). In most traditional analyses it is assumed that the verb in root sentences has been "moved" from the final position to the second position. Koster (1975) argues for this analysis of Dutch. Thus, a simple root sentence in German and Dutch is analyzed as in the following examples:

Vandaag kust$_i$ de man de vrouw, ϵ_i
Today kisses the man the woman

Vandaag heeft$_i$ de man de vrouw ϵ_i gekust
Today has the man the woman kissed

Vandaag [ziet en hoort]$_i$ de man de vrouw ϵ_i
Today sees and hears the man the woman

In DCG such an analysis can easily be defined by unifying the information on the verb in second position to some empty verb in final position, as exemplified by the simple grammar for a Dutch fragment in Figure 7. In this grammar, a special empty element is defined corresponding to the missing verb. All information on the verb in second position is percolated through the rules to this empty verb. Therefore the definition of the several VP rules is valid for both root and subordinate clauses.[7] The problem comes about because the generator can (and must) at some point predict the empty verb as the pivot of the construction. However, in the definition of this empty verb no information (such as the list of complements) will get instantiated. Therefore, the VP complement rule (11) can be applied an unbounded number of times. The length of the lists of complements now is not known in advance, and the generator will not terminate.

Van Noord (1989a) proposes an ad hoc solution that assumes that the empty verb is an inflectional variant of a verb. As inflection rules are delayed, the generation process acts as if the empty verb is an ordinary verb, thereby circumventing the problem. However, this solution only works if the head that is displaced is always lexical. This is not the case in general. In Dutch the verb second position can not only be filled by lexical verbs but also by a conjunction of verbs. Similarly, Spanish clause structure can be analyzed by assuming the "movement" of complex verbal constructions to the second position. Finally, in German it is possible to topicalize a verbal head.

```
s2/Sem ---> adv(Arg)/Sem, s1/Arg.

s1/Sem ---> v(A,B,nil)/V, s0(v(A,B)/V)/Sem.

s0(V)/Sem ---> np/Np, vp(np/Np,[],V)/Sem.

vp(Subj,T,V)/LF --->
  np/H, vp(Subj,[np/H|T],V)/LF.                          (11)

vp(A,B,C)/D ---> v(A,B,C)/D.

vp(A,B,C)/Sem ---> adv(Arg)/Sem, vp(A,B,C)/Arg.          (12)

v(A,B,v(A,B)/Sem)/Sem ---> [].

np/john ---> [john].

np/mary ---> [mary].

adv(Arg)/today(Arg) ---> [vandaag].

v(np/S,[np/O],nil)/kisses(S,O) ---> [kust].
```

Figure 7 Dutch Grammar Fragment.

Note that in these problematic cases the head that lacks sufficient information (the empty verb anaphor) is overtly realized in a position where there is enough information (the antecedent). Thus it appears that the problem might be solved if the antecedent is generated *before* the anaphor. This is the case if the antecedent is the semantic head of the clause; the anaphor will then be instantiated via top-down information through the `chained_nodes` predicate. However, in the example grammar the antecedent is not necessarily the semantic head of the clause because of the VP modifier rule (12).

Typically, there is a relation between the empty anaphor and some antecedent expressed implicitly in the grammar; in the case at hand, it comes about by percolating the information through different rules from the antecedent to the anaphor. We propose to make this relation explicit by defining an empty head with a Prolog clause using the predicate `head_gap`.

```
head_gap(v(A,B,nil)/Sem,
    v(A,B,v(A,B)/Sem)/Sem).
```

Such a definition can intuitively be understood as follows: once there is some node X (the first argument of `head_gap`), then there could just as well have been the *empty* node Y (the second argument of `head_gap`). Note that a lot of information is shared between the two nodes, thereby making the relation between anaphor and antecedent explicit. Such rules can be incorporated in the generator by adding the following clause for *connect:*

```
connect(Pivot,Root) :-
  head_gap(Pivot,Gap), connect(Gap,Root).
```

Note that the problem is now solved because the gap will only be selected after its antecedent has been built. Some parts of this antecedent are then unified with some parts of the gap. The subcategorization list, for example, will thus be instantiated in time.

5 Further Research

We mentioned earlier that, although the algorithm as stated is applicable specifically to generation, we expect that it could be thought of as an instance of a uniform architecture for parsing and generation, as the Earley generation algorithm was. Two pieces of evidence point this way.

First, Martin Kay (1990) has developed a parsing algorithm that seems to be the parsing correlate to the generation algorithm presented here. Its existence might point the way toward a uniform architecture.

Second, one of us (van Noord 1989b) has developed a general proof procedure for Horn clauses that can serve as a skeleton for both a semantic-head-driven generator and a left-corner parser. However, the parameterization is much more broad than for the uniform Earley architecture (Shieber 1988).

Further enhancements to the algorithm are envisioned. First, any system making use of a tabular link predicate over complex nonterminals (like the `chained_nodes` predicate used by the generation algorithm and including the link predicate used in the BUP parser; Matsumoto et al. 1983) is subject to a problem of spurious redundancy in processing if the elements in the link table are not mutually exclusive. For instance, a single chain rule might be considered to be applicable twice because of the nondeterminism of the call to `chained_nodes`. This general problem has to date received little attention, and no satisfactory solution is found in the logic grammar literature.

More generally, the backtracking regimen of our implementation of the algorithm may lead to recomputation of results. Again, this is a general property of backtrack methods and is not particular to our application. The use of dynamic programming techniques, as in chart parsing, would be an appropriate augmentation to the implementation of the algorithm. Happily, such an augmentation would serve to eliminate the redundancy caused by the linking relation as well.

Finally, to incorporate a general facility for auxiliary conditions in rules, some sort of delayed evaluation triggered by appropriate instantiation (e.g. wait declarations; Naish 1986) would be desirable, as mentioned in Section 3.4. None of these changes, however, constitutes restructuring of the algorithm; rather, they modify its realization in significant and important ways.

Acknowledgments

The research reported herein was primarily completed while Shieber and Pereira were at the Artificial Intelligence Center, SRI International. They and Moore were supported in this work by a contract with the Nippon Telephone and Telegraph Corporation and by a gift from the Systems Development Foundation as part of a coordinated research effort with the Center for the Study of Language and Information, Stanford University; van Noord was supported by the European Community and the Nederlands Bureau voor Bibliotheekwezen en Informatieverzorgin through the Eurotra project. We would like to thank Mary Dalrymple and Louis des Tombe for their helpful discussions regarding this work, the Artificial Intelligence Center for their support of the research, and the participants in the MiMo2 project, a research machine translation project of some members of Eurotra-Utrecht.

References

Calder, J.; Reape, M.; and Zeevat, H. 1989 "An Algorithm for Generation in Unification Categorial Grammar." In *Proceedings of the 4th Conference of the European Chapter of the Association for Computational Linguistics,* 233–240.

Colmerauer, A. 1982 PROLOG II: Manuel de Réference et Modèle Théorique. Technical report, Groupe de'Intelligence Artificielle, Faculté des Sciences de Luminy, Marseille, France.

Cooper, R. 1983 "Quantification and Syntactic Theory," Volume 21 of *Synthese Language Library.* D. Reidel, Dordrecht, the Netherlands.

Dymetman, M. and Isabelle, P. 1988 "Reversible Logic Grammars for Machine Translation." In *Proceedings of the Second International Conference on Theoretical and Methodological Issues in Machine Translation of Natural Languages.*

Evers, A. 1975. *The Transformational Cycle in German and Dutch.* Ph.D. Thesis, University of Utrecht, Utrecht, the Netherlands.

Fodor, J. D. In press. "Cross Serial Dependencies and Subcategorization Percolation." In R. Rieber (ed.), *CUNYForum,* Volume 15. City University of New York, New York.

Halvorsen, P.-K. and Kaplan, R.M. 1988 "Projections and Semantic Description in Lexical-Functional Grammar." In *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, Japan, 1116–1122.

Hirsh, S. 1987 "P-PATR, a Compiler for Unification Based Grammars," In V. Dahl and P. Saint-Dizier (eds.), *Natural Language Understanding and Logic Programming, II.* Elsevier Science Publishers, New York, NY: 63–78.

Hobbs, J.R. and Shieber, S.M. 1987 An Algorithm for Generating Quantifier Scopings." *Computational Linguistics,* 13:47–63.

Huybrechts, R.A.C. 1984 "The Weak Inadequacy of Context-Free Phrase Structure Grammars," In G. de Haan, M. Trommelen, and W. Zonneveld (eds.), *Van Periferie naar Kern.* Foris, Dordrecht, the Netherlands.

Kay, M. 1990 "Head-Driven Parsing." In M. Tomita (ed.), *Current Issues in Parsing Technology.* Klumer Academic Publishers, Dordrecht, the Netherlands.

Koster, J. 1975 "Dutch as an SOV Language." *Linguistic Analysis,* 1:(2):111–136.

Matsumoto, Y.; Tanaka, H.; Hirakawa, H.; Miyoshi, H.; and Yasukawa, H. 1983 "BUP: A Bottom-Up Parser Embedded in Prolog." *New Generation Computing,* 1(2):145–158.

Moortgat, M. 1984 "A Fregean Restriction on Meta-Rules" In *Proceedings of New England Linguistic Society,* 14:306–325.

Naish, L. 1986 "Negation and Control in Prolog," Volume 238 of *Lecture Notes in Computer Science.* Springer-Verlag Berlin, F.R.G.

van Noord, G. 1989a "BUG: A Directed Bottom-Up Generator for Unification Based Formalisms." *Working Papers in Natural Language Processing* 4, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht, Utrecht, the Netherlands.

van Noord, G. 1989b "An Overview of Head-Driven Bottom-Up Generation." In *Proceedings of the Second European Workshop on Natural Language Generation,* Edinburgh, Scotland.

Pereira, F.C.N. and Shieber, S.M. 1985 "Prolog and Natural-Language Analysis," Volume 10 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, CA.

Pereira, F.C.N. and Warren, D.H.D. 1983 "Parsing as Deduction." In *Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics,* 137–144.

Pereira, F.C.N. 1981 "Extraposition Grammars." *Computational Linguistics,* 7(4):243–256.

Pollard, C. 1988 "Categorial Grammar and Phrase Structure Grammar: An Excursion on the Syntax-Semantics Frontier," In R. Oehrle, E. Bach, and D. Wheeler (eds.), *Categorial Grammars and Natural Language Structures.* D. Reidel, Dordrecht, the Netherlands.

Shieber, S.M. 1985a "An Introduction to Unification-Based Approaches to Grammar," Volume 4 of *CSLI Lecture Notes.* Center for the Study of Language and Information, Stanford, CA.

Shieber, S.M. 1985b "Using Restriction to Extend Parsing Algorithms for Complex-Feature-Based Formalisms." In *Proceedings of the 23rd Annual Meeting of the Association for Computational Linguistics,* 145–152.

Shieber, S.M. 1988 "A Uniform Architecture for Parsing and Generation." In *Proceedings of the 12th International Conference on Computational Linguistics,* 614–619.

Steedman, M. 1985 "Dependency and Coordination in the Grammar of Dutch and English." *Language,* 61(3):523–568.

Strzalkowski, T. 1989 *Automated Inversion of a Unification Parser into a Unification Generator.* Technical Report 465, Department of Computer Science, New York University, New York.

Wedekind, J. 1988 "Generation as Structure Driven Derivation." In *Proceedings of the 12th International Conference on Computational Linguistics,* 732–737.

Notes

1. See for instance the text by Pereira and Shieber (1985) for an overview and further references.
2. Again, see the text by Pereira and Shieber (1985, p. 172ff.) and references therein.
3. Pereira and Warren (1983) point out that Earley deduction is not restricted to a left-to-right expansion of goals, but this suggestion was not followed up with a specific algorithm addressing the problems discussed here.
4. We use the term "analysis tree" rather than the more familiar "parse tree" to make clear that the source of the tree is not necessarily a parsing process; rather the tree serves only to codify a particular analysis of the structure of the string.
5. Further details of the use of shuffle in scoping are given by Pereira and Shieber (1985).
6. This compels us to represent logical form bound variables as Prolog constants, in contrast to the standard practice in logic grammars.
7. For simplicity the grammar does not handle topicalization, but (counterfactually) assumes that the topic is some adverbial constituent. Topicalization can be handled by gap-threading (Pereira 1981; Pereira and Shieber 1985).